Evolution and Reuse of Orthogonal Architecture

Václav Rajlich, *Member, IEEE Computer Society*, and João H. Silva

Abstract—In this paper, we present a case study of evolution (or vertical reuse) in the domain of visual interactive software tools. We introduce an architecture suitable for this purpose, called orthogonal architecture. The paper describes the architecture itself, the reverse engineering process by which it was obtained, and the forward engineering process by which it was evolved.

Index Terms—Software reuse, object-oriented programming, orthogonal architecture, software tools, layers, threads, program families, reverse engineering, software evolution.

1 Introduction

SOFTWARE evolution is a process where the program is adjusted to satisfy a new set of requirements. In this respect, software evolution, reuse in vertical domain, and perfective maintenance overlap to a large extent. In this paper, we present a case study of evolution (or vertical reuse) in the domain of visual interactive software tools.

The concepts of vertical reuse and evolution are related to the concept of "program families" [7], [8]. Program families are sets of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analyzing individual differences. The concept of program families is in turn related to the concept of application domain [1], [6], [10]. Software architectures were investigated in several recent papers and reports. In [9], notions and classifications are introduced, and the possibilities of use and reuse of the architectures are discussed. In [2], [4], the well-known concept of layered systems is discussed. In order to fulfill the requirements of evolution, the architecture should possess the following attributes:

1) *Understandability*. Users who have some knowledge of the domain are capable of understanding what each class does in terms of the domain, just by reading the class specifications.
2) *Adaptability*. The classes are capable of evolving, i.e., they are capable of absorbing necessary changes while the overall structure of the system remains unchanged.
3) *Reusability*. The architecture can be transformed from one set of specifications to another one within the domain.

In order to accomplish these objectives, we developed "orthogonal architecture," and a methodology for transforming one program having that architecture into another, satisfying a new set of requirements. In order to confirm both architecture and methodology, we conducted a case study of the domain of visual interactive tools. Our previous experience in this domain is summarized in [11], [12]. In Section 2, we describe the ideal orthogonal architecture. In Section 3, we describe the actual architecture of the Environment for Decomposition and Generalization (EDG). Sections 4 and 5 describe processes of reverse and forward engineering related to EDG architecture, respectively. Section 6 lists the conclusions of this study.

2 Orthogonal Architecture

Orthogonal architecture consists of classes (objects) organized into layers and threads. Layers are a well-known concept, and they are sets of classes on the same level of abstraction [2], [4]. Threads are special cases of subsystems [5], and they consist of classes implementing the same functionality. Threads also can be characterized as sets of classes related to each other by relationship of "use," while there is no "use" relationship among the classes of a layer. The architecture is fully orthogonal if threads are independent of each other, i.e., there is no use between the classes of different threads. The orthogonal architecture has a shared top layer, which invokes individual threads, and a shared bottom layer, which contains classes jointly used by all threads, usually data shared among the threads. Each thread implements a part of requirements specifications, and is independent of other threads.

During the evolution, the requirements change. With orthogonal architecture, each change affects some threads only, leaving the rest intact. Hence the change in requirements is localized, causes a lesser impact, and it is easier to implement. Fig. 1 contains a idealized scheme of orthogonal architecture with four layers and six threads. Black dots represent classes, and lines represent use relationships between them. Specific orthogonal architectures may have different numbers of threads and layers.

The advantages of orthogonal architecture are in the above mentioned properties of understandability, adaptability, and

• V. Rajlich is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: rajlich@cs.wayne.edu.
• J.H. Silva is with the Ford Motor Company, 19540 Allen Rd., Melvindale, MI 48122. E-mail: jsilva6@ford.com.

Manuscript received February 1994; revised March 1995.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number S96002.

0098-5589/96$05.00 ©1996 IEEE

Authorized licensed use limited to: Wayne State University. Downloaded on July 27, 2009 at 16:41 from IEEE Xplore. Restrictions apply.

reusability. The orthogonal grid facilitates understanding; the position of the class in the architecture already says what level of abstraction the class implements, and what functionality it implements. The architecture facilitates reuse because it is shared by all programs of the domain, which have the same layers but may have different threads. And it is easily adaptable, since the threads are independent of each other and hence modification or adaptation of one thread does not affect the other threads. While adapting one thread to new requirements, the programmer needs not to be concerned that he/she will break the other threads.

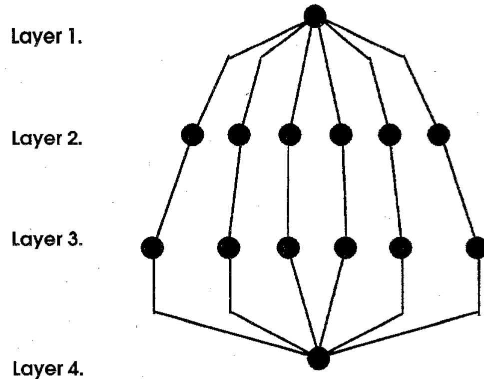


Fig. 1. Orthogonal architecture.

In practice, we allow deviations from this rigid grid, in a similar style in which Broadway is a deviation from the orthogonal street grid in Manhattan. It is a well-known fact that some systems cannot be layered, or that the cost of full layering would be prohibitive [4]. Similarly, the complete separation among the threads may be impossible or its cost may be prohibitive. However, orthogonal architecture serves as an important design ideal. The closer the actual design approximates the orthogonal architecture, the better it supports attributes of understandability, adaptability, and reusability. Every deviation and every irregularity in the architecture weakens the above mentioned properties. All extra costs required by the orthogonal architecture are then weighted against the advantages of higher understandability, adaptability, and reusability.

Not all domains, however, are likely to be suitable for this kind of architecture. In [4], it was remarked that even a less restricted layered architecture would not be suitable for all domains. We believe that the architecture close to that of Fig. 1 is suitable for many interactive repository systems, where the user selects various commands from a menu. The main functionality of the commands is to update or query the shared data repository. (Our notion of "command" is equivalent to "use cases" of [5]). If the domain displays these characteristics, we believe it is suitable for orthogonal architecture. Our case study deals with one such domain, the domain of interactive software tools.

Reuse of an orthogonal architecture is accomplished by a process which is a variant of OODG [13] and OOSE [5]. While layer 1 controls the program, commands are always implemented in the threads. A new program of the same domain can reuse some of the old commands completely, others with modifications, others have to be implemented from scratch, and some are no longer used. The designer maps new commands on the old ones. Then the threads which consist of commands no longer used are discarded, the threads consisting of fully reused commands are kept without change, and the rest of the threads are modified.

The next step consists of modification of the classes in the threads. The process is top-down, first dealing with the top classes of the thread, then with the classes used by them, etc. For each class, class members are modified. Four different scenarios can occur:

- The class member is reused "as is." No changes are required.
- The class member needs to be adjusted to conform to a new set of requirements.
- The class member does not exist, and has to be implemented from scratch.
- The class member is no longer used and is discarded.

Through this process, the orthogonal architecture is adapted step-by-step to the new set of requirements.

3 Domain of Interactive Software Tools

The specific architecture which we describe in this paper is EDG (the Environment for Decomposition and Generalization—pronounced "edge"). EDG supports the OODG methodology of design [13], and its threads and commands are listed in Appendix A. Our goal was not to show that the architecture of EDG is the best possible, but that it possesses the attributes of understandability, adaptability, and reusability as required in the process of evolution. EDG has six layers and six threads, see Fig. 2. It approximates the ideal orthogonal architecture; there are several deviations and irregularities which are caused by variability in the structure of threads, and also the need to share some classes between threads.

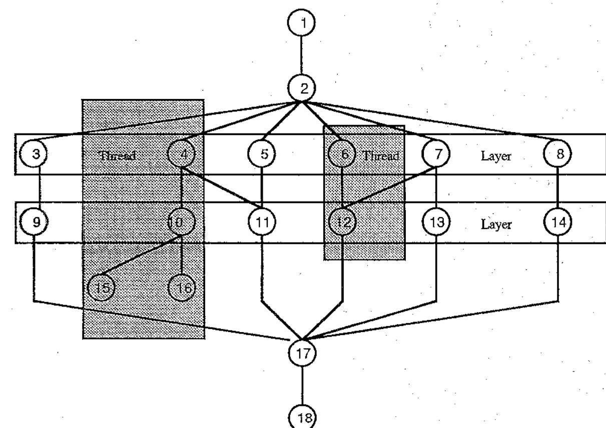


Fig. 2. Architecture of EDG.

In layer 1, function "main" controls the program. Layer 2 implements the menu interface, and supports selection of the commands by the user. Layer 3 encapsulates the user dialogue for all commands. For example, if the command is

Authorized licensed use limited to: Wayne State University. Downloaded on July 27, 2009 at 16:41 from IEEE Xplore. Restrictions apply.

"Save As," the user is asked to enter a file name. Then if the file name is valid, the file is saved.

The actual functionality of the commands was defined at layer 4. Some of the classes which were defined at this layer include: a class to draw graphs on canvas, a class which scans C++ source code and extracts dependencies between objects, etc. Finally, most commands in EDG need the services of a database. The fifth and sixth layers perform these services. Layer 5 encapsulates the specific data model and database interface, while layer 6 is the database itself. They are both custom made specifically for EDG. The database is resident in the main memory and is stored on the disk in a flat file. Efficiency was not a concern, and hence relatively simple data representations and search algorithms were employed.

The threads of EDG are:

- *Project,* which supports commands that operate on entire projects and its files.
- *Graph,* which supports graph display and editing.
- *Views,* which selects information to be presented in a window.
- *Browser,* which supports navigation through the database.
- *Analyzer,* which extracts architectural information from programs.
- *Run,* which interfaces EDG with other tools (compilers, debuggers, make).

The process we employed in our case study consisted of two phases: Reverse engineering where we created EDG from earlier projects, and then forward engineering where we adapted EDG to a new set of requirements.

4 Reverse Engineering: Creating EDG

EDG was created from earlier projects, which had parts of EDG but did not possess the orthogonal architecture described above. Reverse engineering was applied to code written partially in C and partially in C++. The effort involved domain analysis, which was done for the whole domain of visual interactive tools. In domain analysis, we followed a similar process to the one described in [3], and created a list of main objects and commands in the domain of visual interactive tools.

That was followed by a domain design, where the threads, layers, and individual classes of EDG were defined. The orthogonal architecture proved to be a key concept in this phase, and it provided clear guidelines for the individual design decisions. Finally the existing code was analyzed and reengineered into the new code, fitting the EDG architecture. The classes of the new code fall into the following categories: Classes transferred from the previous projects with modifications and classes written from scratch. There were no classes which could be reused from the previous project without any change.

Of the 11,558 lines of the code of EDG, a total of 4,360 belongs to the classes written from the scratch. These classes belong mostly to top and bottom layers, where the impact of the new architecture was most felt. The rest belong to the classes that were modified to a larger or lesser degree. As far as the effort is concerned, the effort to reengineer old code into EDG was approximately 40% of the estimated effort it would take to implement EDG from scratch. For more detailed numbers, see [15].

5 Evolution: From EDG to EDFD

The adaptability of the orthogonal architecture of EDG was tested in the next phase where it was evolved into a different visual interactive tool, Environment for Data Flow Diagrams (EDFD), which partially supports the methodology described in [14]. Appendices A and B contain the list of commands for both EDG and EDFD, and are indicative of the extent of the change done in this process.

In the first step, we mapped the set of new requirements on the pre-existing threads. For example, the command "Hide Object" is mapped on the thread "graph." After all requirements for the new system are assigned to threads, all unnecessary threads are removed from the system. In our case EDFD required three threads: the Data-Flow Editor, the Data Dictionary Editor, and the Defining Functions Editor. The Data-Flow Editor was mapped on the Graph Editor thread of EDG, while both remaining threads were mapped on the Project thread of EDG. Hence the Project thread is modified in two different ways: once to serve as Data Dictionary thread, and a second time as Defining Functions thread. The rest of the threads of EDG were no longer needed, and therefore were discarded.

The next step consists of the modification of the classes in the threads. The process was top-down, first dealing with the top classes of the thread, then with the classes used by them, etc. When modifying a class, we mapped the new specifications of the class on the old one. For "Add an Object," we identified the set of class members in "graph" associated with that command. Three different scenarios have occurred:

1) The class member can be reused "as is." No changes were required.
2) The class member needs to be adjusted to conform to a new set of requirements.
3) The class member does not exist, and has to be implemented from scratch.

The whole architecture was systematically scanned through this process, and adapted for the new set of requirements. The bulk of the changes—not surprisingly—appeared in layer 4, where the functionality of the system resides. The following statistics on the extent of modifications were gathered:

Layer (%)	1	2	3	4	5	6
used as is	100	74.2	94.6	73.5	88.3	78.4
modified		25.8	5.4	2.2	3.8	2.3
implemented from scratch				24.3	7.9	19.3

The total size of EDFD is 4,606 lines. The total effort of the reuse represents 37% of the estimated effort to build the system from scratch. For more detailed numbers, see [15].

Authorized licensed use limited to: Wayne State University. Downloaded on July 27, 2009 at 16:41 from IEEE Xplore. Restrictions apply.

6 Conclusions

We found that building the "perfect architecture" is a step by step process, very similar to exploratory programming. As we coped with the new requirements of EDFD, we were able to perfect the existing architecture of EDG and improve its classes, thus making the architecture more universal and adaptable. In hindsight, we would be able to improve the EDG architecture even further, making it more orthogonal by removing some of the irregularities, simplifying some commands, and merging the threads of "browser" and "views."

We developed a process of three steps to adapt the architecture to a new set of requirements. The process is domain independent. We conjecture that the orthogonal architecture and corresponding methodology for adaptation can be developed for domains other than visual interactive tools. We conjecture that it is applicable to many small and medium sized interactive repository systems, i.e., the systems which react to user's commands and the commands mainly update or query a data repository.

In our case study, we found encouraging productivity figures. In order to reengineer a general architecture into an orthogonal one, we spent approximately 40% of the time compared to implementation from the scratch. In order to adapt this architecture to a new set of requirements, we spent approximately 37% of the time compared to building the system from the scratch. We find these preliminary figures to be very encouraging.

Appendix A – EDG Requirements

EDG is an experimental software tool for the development of C++ programs. It supports the Object-Oriented Decomposition and Generalization methodology [13]. C++ programs can be displayed and edited in two forms: class diagrams and code. Six threads of EDG are described below:

A.1 The Project Manager

The project manager thread implements the commands which operate on entire projects. The commands are:

Open:	Opens an existing project or starts a new project.
Add File:	Adds a new file to the project.
Remove File:	Removes a file from the project.
Save:	Saves current project.
Save As:	Saves current project under a new name.
Quit:	Stops EDG, returns to the operating system.

A.2 The Graph Editor

Graph commands change a program under development, represented as a graph. The commands are:

New:	Starts a new graph editor window and initializes the graph database.
Add	Adds objects to the graph database.
Delete:	Removes objects from the database.
Rename:	Changes the name of an object.
Select:	Changes the active object.
Color:	Changes colors of objects.
Hide:	Removes an object and associated dependencies from a graph. These objects are not displayed but are kept in the database.
Move:	Changes object positions in a displayed graph.
Load Graph:	Reads the information from a file which contains the description of a graph and displays it.
Save Graph:	Writes the graph description into a file.

A.3 The View Manager

The view manager controls the presentation of the information in a window. Results of a query are either graphs drawn in graph windows or text drawn in text window. The commands are:

New:	Creates a new window.
Load Graph:	Reads and displays the information from a file which contains the description of a graph.
Save Graph:	Writes the graph description in a named file.
Show File:	Displays a named file in a text window.
Text Editor:	Starts text editor for a selected file.
Print:	Sends a file to the printer.

A.4 The Code Analyzer

This thread extracts dependencies of C++ programs. The analyzer uses a parser to scan C++ code files and identify C++ code components and their dependencies. The thread supports the following commands:

New:	Creates a new instance of database.
Analyze:	Activate the parser, and populate the analyzer database.

A.5 The Browser

The browser provides information about the set of classes and files comprising the system and dependencies among them. The user may "browse" through the system based on dependencies among classes, files, and identifiers. The commands are:

New:	Creates a new instance of the browser and initialize the browser database.
Select Class:	Changes the current "active class."
Class Info:	Displays information about the "active class," including all dependencies.
Declarations:	The declaration of the active class is displayed.
Definitions:	The definition of the active class is displayed.
Show Info:	This operation is used to give a summary of information about an identifier. Identifiers may be classes or class members.

A.6 The Run Project

The run project contains the set of commands used to interface EDG with other tools (e.g., compilers, debuggers, and code generators). The commands are:

Generate:	Generates C++ code skeletons for classes.
Make:	Generates a Makefile for an entire project based on project descriptions.
Compile:	Executes the make file.
Execute:	Executes the object code for the project.
Debug:	Invokes a symbolic debugger—dbx.

Authorized licensed use limited to: Wayne State University. Downloaded on July 27, 2009 at 16:41 from IEEE Xplore. Restrictions apply.

APPENDIX B – EDFD REQUIREMENTS

EDFD supports the data-flow part of the methodology of [14]. The following are the threads of EDFD:

B.1 The Data Flow Diagram Editor

Data flow diagrams depict a system from the data point of view. Using data flow diagrams, the analyst is able to show how data flows in a system, how data is transformed by the system, and where to store data in the system. The data flow editor supports the following commands:

New:	Starts a new graph editor window and initializes the data flow diagram database.
Add:	Adds objects to the graph database.
Delete:	Removes objects from the database.
Rename:	Changes the name of an object.
Select:	Changes the active object.
Color:	Changes colors of objects in a specific data flow diagram.
Hide:	Removes an object and its associations from the data flow diagram. These objects are not displayed, but are kept in the database.
Move:	Changes object position in the display.
Load Graph:	Reads the information from a file which contains the description of a data flow diagram, and displays it.
Save Graph:	Writes the data flow diagram description into a file.

B.2 The Data Dictionary Editor

The data dictionary defines the meaning of each data flow element. To support this activity we provide the following commands:

New:	Starts a new dictionary.
Add Entry:	Inserts a new object in the data dictionary database.
Delete Entry:	Removes an existing object from the data dictionary database.
Load DD:	Loads a specified data dictionary into main storage
Show DD:	Displays the data dictionary in a text window.
Save DD:	Stores an existing data dictionary in a file for future retrieval.
Print DD:	Prints data dictionary.

B.3 The Functions Editor

The functions from data flow diagrams are described by a text, which may be structured English, pseudocode, or other textual information. The following commands are supported:

New:	Starts a function description.
Add Function:	Inserts a new function in the database.
Delete Function:	Removes an existing function from the database.
Show Functions:	Displays a specific function in read-only text window.
Load Functions:	Copies a specified set of function descriptions into main storage.
Save Function:	Stores an existing set of function descriptions in the database.
Print Function:	Prints the current function.
Print All:	Prints all functions.

REFERENCES

[1] G. Arango, J. Hoskins, and E. Schoen, "Product modelling for software re-engineering." *Proc. 13th Int'l Conf. Software Engineering*, pp. 14-17, Austin, Texas, May 13-17, 1991.

[2] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. Software Engineering and Methodology*, vol. 1, no. 4, pp. 355-398, Oct. 1992.

[3] M.F. Dunn and J.C. Knight, "Software reuse in an industrial setting: A case study," IEEE CS Press, CH2982-7/91, pp. 329-337, July 1991.

[4] D. Garlan and M. Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. 1. World Scientific Publishing Co., 1993.

[5] I. Jacobson, *Object Oriented Software Engineering*. Reading, Mass.: Addison-Wesley, 1992.

[6] J.M. Neighbors, "The Draco approach to constructing software from reusable components." *IEEE Trans. Software Engineering*, vol. 10, no. 5, pp. 564-74, Sept. 1984.

[7] D.L. Parnas, "On the design and development of program families," *IEEE Trans. Software Engineering*, vol. 2, no. 1, pp. 1-9, Mar. 1976.

[8] D.L. Parnas, P.C. Clemens, and D.M. Weiss, "The Modular structure of complex systems." *IEEE Trans. Software Engineering*, vol. 11, no. 3, pp. 259-266, Mar. 1985.

[9] D.E. Perry and A.L. Wolf, "Foundations for the study of software architecture," *Software Engineering Notes*, pp. 40-52, Oct. 1992.

[10] R. Prieto-Diaz, "A domain analysis: An introduction," *ACM Software Engineering Notes*, vol. 15, no. 2, pp. 47-54.

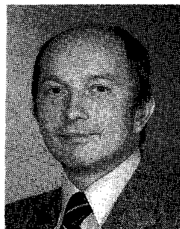
[11] V. Rajlich, N. Damaskinos, W. Korshid, P. Linos, and J. Silva, "An environment for maintaining C programs." *CASE '88 Second Int'l Workshop Computer-Aided Software Engineering*, Cambridge, Mass., July 12-15, 1988.

[12] V. Rajlich, N. Damaskinos, W. Korshid, and P. Linos, "VIFOR: A tool for software maintenance," *Software–Practice and Experience*, vol. 20, no. 1, pp. 67-73, 1990.

[13] V. Rajlich, "Decomposition/generalization methodology for object-oriented programming," *J. Systems and Software*, pp. 181-186, Feb. 1994.

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, N.J.: Prentice Hall, 1991.

[15] J. Silva, "Vertical reuse in software tools: A case study," PhD dissertation, Dept. of Computer Science, Wayne State Univ., Detroit, 1993.



Václav Rajlich received the PhD in mathematics (formal languages and automata) from Case Western Reserve University in 1971. He is now a professor of computer science at Wayne State University and a former chair of computer science (1985-1990). In 1982-1985, he was an associate professor of computer and communication science at the University of Michigan. In 1971-1980, he was a research scientist and later a manager at Research Institute for Mathematical Machines, Prague, Czechoslovakia. He is an author or coauthor of numerous articles on software engineering. His current research interests include design methodologies for object-oriented programs, software comprehension, and software evolution. He has been a program co-chair, general chair, and steering committee chair of the IEEE International Conference on Software Maintenance, and was a general chair of the IEEE Workshop on Program Comprehension. He is a member of the IEEE Computer Society and the ACM.

João H. Silva received the BSE (Licenciatura) degree from the University of Porto, Portugal, in 1981, and MS and PhD degrees in computer science from Wayne State University in 1988 and 1993, respectively. His current research interests include software architectures, vertical software reuse, and tools for real-time embedded systems. He is currently employed by the Ford Motor Company in the Advanced Microcontroller group.

Authorized licensed use limited to: Wayne State University. Downloaded on July 27, 2009 at 16:41 from IEEE Xplore. Restrictions apply.